AMENDMENTS TO THE SPECIFICATION

Page 3, paragraph [0009]:

The number of streams available to a program is regulated by three quantities: the stream limit (**slim**), the current number of streams (**scur**), and the number of reserved streams (**sres**) associated with each protection domain. The current numbers of streams executing in the protection domain is indicated by **scur**; it is incremented when a stream is created and decremented when a stream quits. A create can only succeed when the incremented **scur** does not exceed **sres**, the number of streams reserved in the protection domain. The operations for creating, quitting, and reserving streams are unprivileged. Several streams can be reserved simultaneously. The stream limit **slim** is an operating system limit on the number of streams the protection domain can reserve.

Pages 17-18, paragraph [0061]:

In step 502, the routine fetches and adds to the num_teams variable in the task swap header data structure. In step 503, the routine invokes the tera_team_swapsave_complete operating system call passing the num_streams variable of the team swap header. This operating system call returns immediately when the last team master stream invokes it and returns as its return value a value of 1. For all other team master streams, this operating system call does not return until the task is swapped in. The last team master stream to invoke this operating system is designated as the task master stream. In step 504, if this stream is the task master stream, then the routine continues at step 505, else the routine continues at step ~~506~~507. In step 505, the routine invokes the work_of_final_stream_in_task function. This invoked function does not return until the task is swapped in. Steps 507-521 represent processing that is performed when the task is swapped in. In steps 507-508, the routine fetches and adds a 1 to the

signal_wait variable of the task swap header and waits until that variable equals the num_teams variable in the task swap header. Thus, each team master stream waits until all the other team master streams reach this point in the routine before proceeding. The first stream to increment the signal_wait variable is the task master stream for the swap in. Alternatively, the same stream that was designated as the task master for the swap out can also be the task master for the swap in. In steps 509-514, the routine enables trapping for the domain_signal so that subsequent raising of the domain_signal will cause a trap. The task master stream then processes the Unix signals. During the processing of Unix signals, another domain_signal may be raised. Thus, another swapout can occur before the states of the streams are completely restored. The trap handler handling the domain_signal can handle nested invocations in that the trap handler can be executed again during execution of the trap handler. Therefore, an array of team and swap header data structures is needed to handle this nesting. In step 509, the routine enables the trapping of the domain_signal. In step 510, if this stream is the task master stream, then the routine continues at step 511, else routine continues at step 513. In step 511, the routine invokes the process_signals function to process the Unix signals. In one embodiment, the task master stream creates a thread to handle the Unix signals. In step 512, the routine sets the signal_wait$ synchronization variable of the task swap header to zero, in order to notify the other team master streams that the processing of the Unix signals is complete. In step 513, the routine waits for the notification that the task master stream has processed the Unix signals. In step 514, the routine disables the domain_signal to prevent nested handling of domain_signals. The first save_area data structure in the linked list contains the state of team master stream when the task was swapped out. ~~In step 516, the routine clears the team swap header.~~ In step 515, the routine gets the next save_area data structure from the team swap header. In step 516, the routine clears the team swap header. In steps 517 and 518, the routine fetches and adds a -1 to the num_teams variable in the task swap header and waits until that variable is equal to 0. Thus, each team master stream waits until all other team master streams reach this point in the processing. Thus, these steps implement a synchronization barrier.

One skilled in the art would appreciate that such barriers can be implemented in different ways. In step 519, if this stream is the task master stream, then the routine continues at step 520, else routine continues at step 521. In step 520, the routine clears the task swap header, to initialize it for the next swap out. In step 523, the routine invokes the swap_restart_streams function to restart the slave streams of the team by creating streams, retrieving the save_area data structures, and initializing the created streams. This routine then returns.

Page 21, paragraph [66]:

Figure 10 is a block diagram of data structures used when swapping a task. Each thread has a thread control block 1001 that contains information describing the current state of the thread and points to a team control block 1002 of the team of which the thread is a member. The team control block contains information describing the team and points to a task control block 1005 of the task of which the team is a member. The task control block contains information describing the task. The team control block contains a pointer to a team swap header 1003 that contains information relating to the swapping of the team. The team swap header contains a pointer to a linked list of save_area data structures 1004 that are used to restart the threads when the team is swapped in. The task control block contains a pointer to a task swap header 1006. The task swap header contains information relating to the swapping of the task.

Page 22, paragraph [69]:

When an operating system call that will block is invoked, the operating system (executing on the stream that invoked the operating system call) invokes the rt_return_vp function of the user program. This invocation returns the stream to the user program. The

virtual processor code of the user program can then select another thread to execute on that stream while the operating system call is blocked. Eventually, the operating system call will become unblocked (*e.g.*, the user has finally input data). When the operating system call becomes unblocked, the operating system (executing on one of its own streams) invokes the rt_return_thread function of the user program to notify the user program that the operating system call has now completed. The rt_return_thread function performs the necessary processing to restart (or at least schedule) the thread that was blocked on the operating system call. The rt_return_thread function then invokes the tera_return_stream operating system call to return the stream to the operating system. A malicious user program could decide not to return the stream to the operating system and instead start one of its threads executing on that stream. Thus, a user program could increase the number of streams allocated to it to an amount greater ~~that~~ than the **slim** value set the operating system. The operating system can mitigate the effects of such a malicious user program by not returning any more streams or, alternatively, killing the task when it detects that the user program has failed to return a certain number of the operating system streams.

Page 26, paragraph [75]:

Figure 16A is a diagram illustrating the synchronization of the user program and the operating system when the user program invokes an operating system call that blocks. The diagram illustrates the processing performed by the user stream 1601 and the processing performed by an operating system stream 1602. The solid lines with arrows indicate flow of control from one routine within a stream to another routine within the same stream. The dashed lines indicate the interaction of the synchronization variables. The ellipses indicate omitted steps of the functions. The user program invokes an operating system call by invoking the user_entry_stub routine ~~1000~~1100. That routine in step 1104 invokes the operating system call. As indicated by the solid line between steps 1104 and

1603, the user stream starts executing the operating system call. The operating system call 1603 invokes the rt_return_vp function in step 1604. The rt_return_vp function 1200 stores a value the call_id$ synchronization variable in step 1204, which sets the full/empty bit of the synchronization variable to full. The rt_return_vp function then writes a value into the call_id$ synchronization variable in step 1206. Since the call_id$ synchronization variable just had a value stored in it, its full/empty bit is set to full. This write cannot succeed until the full/empty bit is set to empty. Thus, step 1206 will cause data blocked exception to be raised and the trap handler routine 1500 will be invoked. In step 1501, if the thread is locked, then the trap handler returns to the blocking synchronization write in step 1206. For a locked stream, the process of raising a data blocked exception and returning for a locked thread will continue until the full/empty bit of the call_id$ synchronization variable is set to empty when the operating system call completes. If, however, the thread is not locked, then the trap handler routine places the thread on the blocked pool and executes the virtual processor code to select another thread to execute on that stream. When the operating system call 1605 completes, the operating system in step 1606 invokes the rt_return_thread function 1300 of the user program. This invocation is within a stream allocated to the operating system. The rt_return_thread function 1300 reads the call_id$ synchronization variable in step 1303, which sets its full/empty bit to empty. As indicated by the dashed line, the writing of that synchronization variable in step 1206 then succeeds. The rt_return_vp function then completes the execution of step 1206 and continues to step 1207. In step 1207, the function returns to the location of the user_entry_stub routine immediately after the invocation of the operating system call. The user_entry_stub routine in step 1106 reads the notify_done$ synchronization variable. Since the full/empty bit of this synchronization variable is initially empty, this read blocks. The rt_return_thread routine in step 1305 invokes the tera_return_stream operating system call 1400 to return the stream to the operating system. In step 1403, the tera_return_stream operating system writes a value of 0 to the notify_done$ synchronization variable, which sets its full/empty bit to full. This releases the blocked read in step 1106 and the user_entry_stub routine returns to the user code.

Page 27, paragraph [76]:

Figure 16B illustrates the Upcall Transfer (ut) data structure 1650. The ut data structure 1650 is passed to the operating system when a blocking operating system call is invoked. The ut data structure 1650 contains information ~~in need to~~for ~~synchronize~~ synchronizing the return of the stream to the user program. The was_blocked flag 1655 is set to indicate whether the operating system call was blocked so that the user program can wait until the operating system stream is returned to the operating system and so that the function knows when return values need to be retrieved from the ut data structure 1650. The call_id$ synchronization variable 1660 is used to notify the thread that invoked the operating system call and that has locked the thread, that the operating system call is complete. The notify_done$ synchronization variable 1665 is used to notify the thread that the operating system stream has been returned. The spare_~~ccb~~tcb pointer 1670 points to the spare thread control block that is used when the operating system notifies the user program that the operating system call is complete. The return_value variable 1675 contains the return value of the operating system call.